ASYNCHRONOUS MESSAGE PASSING IN'1'111;J1'1,IU,lG11'1' SYSTEMTESTBED

Scott Burleigh
Jet Propulsion Laboratory
California Institute of Technology
scott@tel.jpl.nasa.gov

ABSTRACT

The flight mission simulation software in the Jet Propulsion Laboratory's Flight System Testbed (FST) is a heterogeneous, d istributed system that is built on an interprocess communication model of asynchronous message passing rather than remote procedure calls (RPCs). The reasoning behind this design decision is discussed; the mechanism used to implement it ("Tramel" --- 'l'ask Remote Asynchronous Message Exchange Layer) is explained; the resulting software architecture is examined; the FST's operational experience with this architecture is reviewed; conclusions are drawn, and possible directions for future work arc considered.

Keywords: message passing, RPC, interprocess communication, distributed processing, parallel processing

1. INTRODUCTION

JPL,'s Flight System Testbed (FST) was developed as a means of simulating deep space flight missions in a controlled and instrumented environment, to aid both in evaluating alternative mission architectures and also in exploiting reusable capabilities to "bootstrap" mission development. As such it includes flight software running on a "~ligllt-like" processor (a single-tmard computer running the VxWorks real-time operat ing system); spacecraft hardware simulat ion software; flight environment simulation so ftware; and control center software that communicates with the flight software through a combination of actual flight communication protocols (CCSDS) and simulated radio signal transmission.

The flight software executing on a computer aboard a deep-space exploration spacecraft and the mission operations software running on computers at a flight operations center on Earth can together be considered a single session of a distributed appl i cat ion, or a *distributed system*. Distributed systems in general may involve an arbitrarily large number of processes executing on any number of computers of possibly varying architecture, running under possibly different operating systems. l'hose computers could be distributed over an arbitrarily broad geographic extent, and each process might be under the control of a different user --- or might be autonomous. The processes don't all begin execution at the same instant, and in fact they might begin and end at arbitrary times. As the individual processes begin and end, the scope and functionality of the application software session as a 1~'hole will grow and shrink. A given process might terminate and be replaced by a process of similar but upgraded capability while the session as a whole is still running (i.e., other processes in the session have not terminated). The session as a whole might not ever terminate at all, since while any of its processes --- on any computers, anywhere --- arc still active it can't be said to have ended. In short: unlike single-program execution sessions, distributed systems can be dynamic, amorphous, constantly evolving, and immortal.

But a given set of processes isn't a distributed application session unless the processes act in concert to solve some application problem (however broadly stated), and this

requires that data flow among them to coordinate their operations. That is, inter-process communication is a prerequisite to distributed application processing.

Inevitably, the character of that processing is substantially dictated by the features and limitations of the inter-process communication mechanism. For example, the potential lifetime of a session is limited if it must be terminated in order to reconfigure its communication pathways.

A deep spat.c flight mission system differs from most distributed applications in that the vast distance between nodes of the network makes the speed of light a significant constraint on the data propagation performance of the software. This constrain{, though, is just an extreme instance of a more general inter-process communication problem, that the round-trip communication rates supported by standard protocols -- -sad therefore the performance of distributed software -- can vary appreciably with variations in such factors as noise and congestion on the links.

A high degree of concurrent or parallel execution in a distributed application largely solves this problem: an increase in round-trip communication latency may increase the time required to complete any single transaction (the Iifc.time of any single thread of processing), but the processing rate of the application as a whole -- the number of transactions begun pcr second, or completed pcr second -- remains constant so long as the number of concurrent threads of execution is allowed to increase. This concurrency also tends to maximize efficient utilization of processing resources.

However, the performance of any single-threaded application process that cannot proceed until a query have been answered (e.g., until a remote procedure call - RPC - has been completed) is unavoidably degraded by impaired link performance. Synchronous interprocess communication - tbc "client/server" model that is widely used in modern distributed software (Refs. 1, 2) - in general limits processing concurrency and therefore the operational performance of distributed applications.

One way to address this defect is to spawn a separate thread of execution whenever an RPC is issued. The thread that issued the RPC is suspended until the RPC is completed, but other threads of the same process can continue and parallelism is preserved. This approach, however, requires that programmers become comfortable with multithreaded applications; the cognitive leverage gained by providing RPCs, which attempt to extend the familiar function invocation programming style into distributed applications development, is offset by the cost of mastering the intricacies of thr cad programming. There arc other drawbacks as well:

- '1'here is as yet no common thread implementation standard that is as widely available, accepted, and understood as, for example, TCP sockets,
- Even the most efficient threading systems must devote appreciable system resources to thread management: stacks, scheduling, etc.
- Data in the address space shared by multiple application threads is shared data. As such, it must be protected from corruption due to overlapping updates; some access serialization dc.vice such as a mutex semaphore is needed. The application programmer must usc that device carefully, avoiding not only race conditions but also deadlock.

An arguably simpler strategy is simply to reject synchronous communication (RPCs) in favor of a portable asynchronous message exchange system equipped with a mechanism for linking reply messages (received asynchronously) to the contexts in which they arc needed. Tramel ('l'ask Remote Asynchronous Message Exchange 1 ayer), developed at JPL for use in the Flight System Testbed, is such a system.

2. TRAMEL

Tramel was designed to provide a capable common platform for distributed computing that could help to insulate mission applications software not only from extreme link performance constraints but also from other development and performance constraints imposed by most other network computing technologies. Asynchronous message passing gave way to RPCs in modern software largely because it was complex to learn, tedious to program, and difficult to implement in a robust fashion. The aim of Tramel's design is to deliver a message-passing system that is more powerful and efficient, and no harder to use, than RPCs.

Tramel insulates application code as much as possible from such inter-process communication details as connection establishment, communication protocol, and differences in processor architecture and operating system. Application software sessions are self-configuring at run time; the order in which processes begin participating in a session is immaterial. Tramel provides a built-in mechanism for linking reply messages, received asynchronously, to the contexts in which they are needed (*projects*, discussed below), enabling processes to converse in a pseudo-synchronous fashion without sacrificing parallel execution. It does so without requiring an OS-supported multithreading system. Where multithreading is available, developers are free to use it as they wish; in its absence, messages are processed sequentially, so access to the process% data is automatically serialized. And although Tramel gains no special leverage from programmers' facility with function calls, its event-loop-based programming model will be familiar to developers of software built on graphical user interface systems such as X Windows.

Finally, Tramel supports an optional publish/subscribe communication model that further shields application code from having to understand the configuration or state of the distributed application session at any time. In effect, each Tramel-speaking process (task) plugs itself into a data "grid", much as producers and consumers of electric power -- say, a hydroelectric plant and a kitchen toaster -- plug into an electric power grid. A Tramel process inserts into the data grid whatever data it produces, without having to know much about the consumer(s) of that data, and it draws from the grid whatever data it requires without having to know much about the producer(s).

2.1 Application Structure

The basis of Tramel is a peer-to-peer direct data transmission system modeled on object-oriented programming concepts: if operating system tasks (threads, processes) can be thought of as coarse-grained "objects", then Tramel can be regarded as a mechanism for message exchange among those objects. Each message is tagged with some application-specific integer, called a *subject*, that identifies the type of the message; message subjects are analogous to "method selectors" in object-oriented programming languages such as Smalltalk. Each message may also optionally have *content*, an arbitrary array of bytes, analogous to the argument(s) of a Smalltalk message.

The general term for a Tramel-speaking task, thread, or process is *node*; each node has a Tramel data structure called a *node state* that manages the data required to support communication between this node and others. A node can send a message to a specified node, refer (relay) a message it has received to some other specified node, subscribe to all published messages having a specified subject, publish a message, or reply to a message.

The Tramel application programming model is based on event-driven processing; most application code is non-compute-intensive and is invoked only from the event loop in response to the detection of system events. (This is similar to the way X Windows applications are normally organized. The Tramel proxy system, a mechanism that enables software which doesn't fit this model to interoperate with Tramel nodes, is discussed later.) Node-specific callback functions are specified for the subjects of all messages the node is prepared to respond to, regardless of whether or not the node has subscribed to those subjects. On receipt of a message, Tramel automatically passes the message to the designated message handling function.

A node can also specify a callback function to be invoked upon receipt of a reply to a given message. Moreover, a node can specify a callback function to be invoked when a specified length of time bas passed since, for example, transmission of a message to which a reply is expected. The Tramel *project* mechanism used to implement both of these capabilities. A project can be thought of as "something a node is doing" that may continue across multiple exchanges of messages and/or timeout expirations, somewhat like an extremely lightweight thread of execution and somewhat like the "long transactions" supported by some object database management systems. Each node may have an arbitrary number of concurrent active projects.

Two kinds of events may occur in the course of a project: receipt of a reply message and expiration of a time limit. Associated with each project is a callback function that Tramel automatically invokes whenever any such project event occurs. Each project may also have a *context*, a user-dcfillcd data object that persists throughout the project's duration and enables the callback functions for a single project to communicate among themselves over time.

The arrival of an original or reply message and the expiration of a timer are three of the types of events'1'ramc] can be used to manage. Tramel also provides facilities for responding to three other types of events: the arrival of data at designated file descriptors not used by Tramel (such as pipes) and two kinds of changes in "message space" configuration -- the registration of a new node of a specified name and the termination ofa specified node.

2..?. Message spaces

A set of nodes, all of which ---and no others --- can potentially exchange messages with one another via Tramel, is a *message space* (see Figure 1); a message space can be thought of as a functional network of processes that is built on top of (but generally ignorant of) some physical network of cables and host machines.

Every message space is a session of the use of some distributed *application*, identified by an application name. Multiple sessions ofa single application (for example, a "production" session and a "test" or "development" session) may exist at the same time. Distinct sessions of the same application are distinguished by the names of the *authorities* that arc responsible for configuring and running them. At tbc same time, a single authority might be responsible for sessions of multiple applications, e.g., there might be a "test" session of a "payroll" application and also a "test" session of a "payables" application. So a message space can be thought of as a section in a matrix of two dimensions, application and authority, and it is uniquely identified by the combination of its application name and its authority name.

The performance of a message space may sometimes be enhanced by partitioning it into multiple *regions*. Message space regions correspond roughly to regions of network geography, such as subnets, and in practice tbcy usually correspond roughly to regions of physical geography; they arc identified by *site* names. Regions of multiple message spaces may, of course, coexist at the same physical/network site.

That is, the "test payroll" message space might have an "albany" region and a "cleveland" region, and the "test payables" message space might also have "albany" and "cleveland" regions, The site names "albany" and "cleveland" identify displacements along a third Tramel dimension, site; any region can be thought of as a section in a matrix of three dimensions -- application, authority, and site -- and it is uniquely identified by the combination of its application name, its authority name, and its site name.

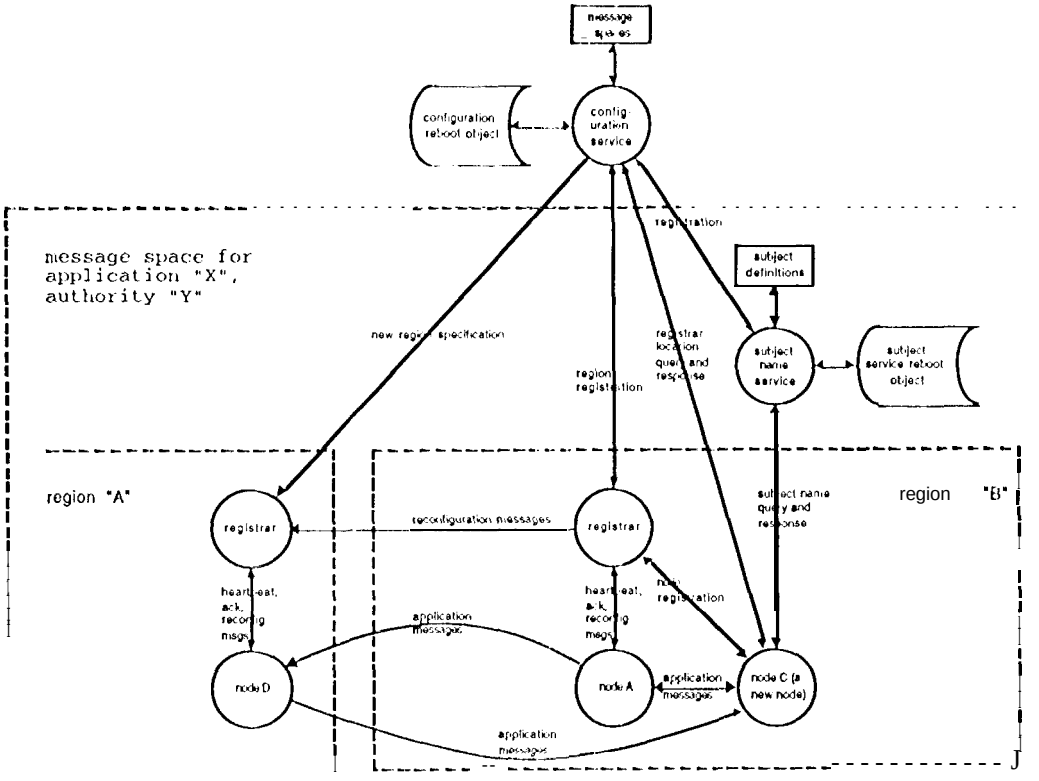

Figure 1: *Tramel functional architecture*

Every message space always comprises at least one region, and each node resides within (is registered in) some region; in the simplest case all nodes of the message space reside in the same region. Each region is served by a registration service daemon, or *registrar*, which is responsible for monitoring the health of all nodes in the region and for propagating four kinds of message space configuration changes: node registrations and terminations, subscriptions, and unsubscriptions. On receipt of one of these reconfiguration messages from a node in its own region, the registrar immediately propagates the message to every other node in the same region anti then to the registrars of all other regions in the message space; on receiving such a message from a remote region's registrar, the registrar propagates it to all nodes in its own region.

The registrars themselves register with a central configuration service daemon at some well-known host and port number. Any Tramel software installation must have at least one configuration service daemon running at all times in order to enable registrars and nodes to register, and all registrars and nodes of the same message space must register through the same configuration server; the registrars and nodes for any number of message spaces may register with the same configuration service daemon.

2.3 Node registration

Each node has a name, an application-specific ASCII string containing no whitespace, which generally indicates its function within the application but need not uniquely identify it within its message space. Each node also has one or more "network identities", one for each underlying communication protocol ("tcp", "fife", or "vxpipe") on which the node is prepared to receive messages. A new node joins a message space by registering itself within some region of the message space, i.e., by announcing its name and its network identities to the region's registrar.

However, knowledge of how to communicate with that registrar can't be built in at compile time because the relevant registrar might be running on different network hosts at different times, For this reason, the first step in registering a new node is to contact the configuration service daemon. (The host name and port number of the configuration service daemon are normally supplied as environment variables.) The configuration server tells the new node how to contact its registrar. The node requests a node number from the registrar and registers. The registrar ensures that all other nodes in the message space learn the new node's name, node id, and network identities. "1'hose nodes in turn announce their own names, node IDs, and network identities to the new node.

To sum up: the detailed configuration of any Tramel message space is developed dynamically and automatically at run time as the members of the message space register themselves. In fact, we can more formally define the term "region" as the set of all nodes that issue their registration, termination, subscription, and unsubscription messages through a given registrar.

2.4 Application scaling

Several aspects of the design of Tramel are aimed at enabling the system to support large and small applications equally well.

- Tramel stores project data and message management rules in arrays that are directly indexed by, respectively, project numbers and subject numbers. This enables Tramel to scale relatively well for large numbers of projects and subjects: the time cost of routing a message to the relevant callback is constant regardless of the size or complexity of the message space.
- Tramel's region partitioning scheme distributes the load of propagating reconfiguration messages among multiple registrars on (normally) multiple hosts, so that reconfiguration is parallelized; adding regions doesn't significantly reduce reconfiguration speed. It also enables a new node to register quickly and to begin immediately interacting with nearby, easily reachable nodes even if some parts of the message space are in other cities, continents, or regions of the solar system.
- Tramel gets much of its portability from its reliance on standard file descriptors as I/O paths and on the standard *select* system call as a mechanism for multiplexing input. However, the number of file descriptors available to an application program is always limited. Connections between nodes consume file descriptors; also they are usually costly to establish. As the number of nodes in an application increases, the number of potentially necessary connections between nodes increases exponentially but the number of available file descriptors is constant. Tramel resolves this conflict by managing connections automatically, breaking one when necessary to release a file descriptor for another one that is about to be accessed. A least-rcccilty-uscci algorithm selects connections to be sacrificed; this keeps the time spent in rc-establishing connections relatively small for applications in which most nodes normally communicate only with a small number of others. (This automatic management of connections also simplifies the use of Tramel in application development, since it entirely hides the connections from the developer.)

All communication between nodes and daemons, and among daemons, is based on TCP/IP socket connections. To minimize the overhead of establishing connections, each registrar keeps connections to every node in its region and to the registrars of all other regions open at all times. This limits both the number of regions in a message space and the number of nodes that can occupy any single region: the sum of the region count and the number of nodes in the region must be less than the total number of file descriptors available to the registrar.

However, even under this constraint the total number of nodes in any single message space can be quite large. For example, a message space running under an operating system that allows each task to have 64 open file descriptors would be limited to around 900 nodes: each registrar could be serving a region of about 30 nodes and connected to about 30 other such registrars. Doubling the number of available file descriptors per task to 128 would quadruple the maximum message space size to about 3600 (60 regions x 60 nodes per region) and so on.

2.5 Heartbeat monitoring

Because the number of nodes in a region is limited, it is important for a registrar always to detect the terminations of nodes in its region so that new nodes can replace the ones that terminated. Normally a node terminates explicitly or the loss of the socket connection to the node indicates its termination. If the host on which a node resides is simply powered off or rebooted, however, TCP itself is terminated on that host and no loss of connection is transmitted to the registrar.

For this reason, nodes automatically send "heartbeat" messages to their registrars at fixed intervals, normally every 20 seconds. The registrar interprets failure to deliver a heartbeat within two heartbeat intervals after the prior heartbeat message as an indication that the node has terminated. Whenever it detects the termination of a node (either an explicit termination or a termination imputed from loss of connection or heartbeat failure), the registrar informs all other nodes in the message space of the node's demise. When the termination is imputed from a heartbeat failure, the registrar also tries to send a message to the terminated node telling it that it has been presumed dead; if this node is in fact still running (perhaps it had hung trying to write on a blocked file descriptor), it terminates immediately on receipt of this message. This ensures that other application behavior that was triggered by the imputed termination will not be invalidated.

2.6 Fault tolerance and automatic reconfiguration

In addition to monitoring the heartbeats of all nodes in its region, each registrar also acknowledges every heartbeat. A node expects the registrar's acknowledgment of the prior heartbeat to arrive before the time at which it must send the next heartbeat message, and it interprets failure to deliver an acknowledgement message by this deadline as an indication that the registrar itself has crashed. When a registrar crash is detected, each node in the dead registrar's region sends a message to the configuration service daemon informing it of the registrar's demise. The configuration service daemon automatically restarts the registrar (possibly on a different host).

These reciprocal monitoring measures make Tramel applications relatively fault

- When a node crashes, its registrar detects the loss of heartbeat within two heartbeat intervals and notifies the rest of the message space. Message transmission everywhere is unaffected.

- When a registrar crashes, its nodes detect the loss of heartbeat acknowledgement within one heartbeat interval and request that the registrar be restarted. During the time that the region has no registrar, transmission of application messages among nodes of the message space is unaffected, but the heartbeat failures of crashed nodes are not detected and reconfiguration messages originating in the region (registrations, terminations, subscriptions, and unsubscriptions) are not propagated to any nodes. However, after the registrar is restarted it will eventually detect the losses of heartbeat from all crashed nodes and issue obituaries to the message space; also, nodes that issued unpropagated reconfiguration message.s will automatically re-issue them (because no registrar ever acknowledged them), eventually bringing the message space back into a comet slate.

Since the maximum heartbeat interval is twenty seconds, within the first sixty seconds after restart the registrar will have received heartbeat messages from every node that is still running in the region and will therefore know accurately the configuration of the region. This accurate configuration information must be delivered to new nodes at the time they start up. For this reason, during the first sixty seconds after the registrar restarts it accepts only connections from existing nodes in the region; if it accepted a connection from a new node before being certain of the status of all old ones, it would run the risk of delivering incorrect information to the new node.

2.7 Symbolic names for subject numbers

Message subjects, as noted above, are integers with application-defined semantics. This minimizes the cost of including subject information (in effect, message type) in every message, and it makes Tramel's internal processing simpler and faster: Tramel records subscription and message handling information in dynamically allocated and possibly sparse arrays that are indexed by subject number. This means, though, that message management control arrays must be large enough to accommodate the largest subject numbers used in the application. The use of extremely large subject numbers will therefore cause these arrays to consume huge amounts of memory. In general, it is best for a Tramel application to use the smallest subject numbers possible, starting with 1.

One way to ensure this is to cite message subjects by symbolic name in application code, rather than cite the subject numbers themselves. This is because the mapping of subject names to numbers, and vice versa, is performed by *subject service* functions which communicate with a subject service daemon.

Each message space can have at most one subject service daemon, which manages a private database of subject definitions for the message space. Each subject definition pairs a subject name with a subject number and, optionally, a message content format string. The daemon itself assigns numbers sequentially (starting at 1) to subject names, in the order in which the subject names are declared to it. An application node declares a subject name by invoking one of the subject service library functions. Once a subject has been declared, other subject service library functions enable all nodes in the message space to determine the. number corresponding to its name or the name corresponding to its number.

In addition to conserving memory, citing subjects by name can also help to reduce error in large system development, can simplify the dynamic definition of new message subjects at run time, and can provide a mechanism to aid in linking multiple message spaces. Note, though, that the use of the Tramel subject service is strictly optional,

2.8 Message transmission reliability

Normally Tramel attempts to guarantee the in-order delivery of every byte of application message data entrusted to it. Although Tramel could be layered on top of virtually any transport-level data communication protocol, this design goal limits the pool of potential implementations to those based on reliable protocols such as TCP/IP, POSIX FIFOs, and VxWorks pipes. Moreover, when application code calls a Tramel message transmission function, control is normally returned to the application only when the message has been completely transmitted --- though not necessarily received --- by the underlying protocol; if the protocol cannot complete the transmission (e.g., the protocol is TCP/IP and the socket connection is blocked), Tramel waits for the channel to be unblocked. That is, Tramel normally assures hat data production is not permitted to outstrip data consumption anywhere in the message space.

1 lowever, in some circumstances this commitment to reliable and orderly data flow may be unnecessary or even undesirable: the application may require that control be returned to it immediately, or it may be permissible to drop some messages. In this case, Tramel configuration functions may be invoked to bead the rules,

Disabling flow control causes Tramel to return control to the application immediately on any message transmission, regardless of whether or not the message has been completely transmit itted. When flow control is disabled and tile initial attempt to transmit a message was only partially successful (part but not all of the message was transmitted), Tramel retains the balance of the message in an internal buffer and transparently attempts to retransmit it whenever the opportunity arises. Tramel never sends partial messages.

Normally, any message that could not be even partially transmitted on first attempt is retained for retransmission in the same way. That is, reliable message delivery is still guaranteed even with flow control disabled. However, tile buffering of undelivered messages may cause Tramel to consume more memory than we're willing to allocate to message transmission. If the undelivered messages are expendable we can further modify the node's message transmission behavior by disabling retransmission as well.

With retransmission disabled, any message that cannot be even partly transmitted is simply discarded. Reliable delivery is sacrificed, but control is immediately rc.turned to the application on any message transmission and potential memory consumption is minimized.

2.9 Notes on the architecture of Tramel

The syntax of the message content format strings mentioned earlier must be defined by a separate presentation layer implementation; it is opaque to, and is not used by, the Tramel subject definition service or Tramel itself. That is, Tramel is a relatively low-level communication system, in that the content of a Tramel message is simply an arbitrarily long array of bytes. In ISO terms, Tramel is session-layer software; message semantics arc left to the application or presentation layer.

Consequently there are no Tramel compiler extensions or application preprocessors for message content marshalling and unmarshalling, as provided by many RPC-based systems. A new program is added to a Tramel application simply by including the Tramel header file, embedding Tramel function invocations in the source code, compiling (using any C compiler), and linking with the Tramel library. Naturally, the application developer may choose to marshal the content of a message in some platform-independent form, e.g., by using XDR. But this is an application decision;

marshalling overhead is incurr cd not by Tramel but by the application, and the nature of the marshaling is left to the developer's discretion.

Every Tramel node is self-coataincel, and all arc peers. There is no central message muter or Tramel communication "run-time". The only daemon processes required to operate a Tramel application arc those discussed above that enable the propagation of message space reconfiguration messages; the only other configuration elements of a Tramel application arc a handful of optional environment variables. '1'here arc no configuration files other than those that arc built automatically by the daemons to enable them to restart correctly. During application message exchange operations, Tramel gets all the information it needs directly from the arguments passed by application code when Tramel library functions arc called.

2.10 integration with other software

Tramel has so far been ported to SunOS 4.1.3 and Solaris 2 on Sparc machines, to AIX on 1{ S6000 workstations, 10 both VxWorks and IRIX5 on Mips hardware, and to VxWorks on Motorola 68040 -based and Powerl'C-based single-board computers. Integration modules have been written to enable software that uses X Windows or is written in 't'cl to participate in a Tramel application universe. A LabView interface is also available, as is an interface that supports rca]-time programming under VxWorks.

In addition, a Tramel "proxy" system has been developed. It enables software that doesn't readily fit the Tramel programming, mode] - in which computationally non-intensive callback functions arc invoked from event loops – to interact with nodes of a Tramel message space. Briefly, a Tramel proxy client invisibly spawns a separate "proxy" process which registers as a message space node. The client participates in the message space indirectly, through a pipe to the proxy. As such, the client itself is exempt from heartbeat cycle discipline and can poll for messages, "sample" the content of the latest messages of specified subject, issue synchronous queries, etc.

3. FST SOFTWARE ARC] HTECTURE

Figure 2 illustrates the architecture of the FST flight simulation software.

"Flight software" -- -software that could, in theory, be installed in a flight computer with little or no modification and used to operate a spacecraft -- is rigorously segregated from "support equipment" -- software that simulates spacecraft hardware or the flight environment. That is, messages arc never passed directly between flight software elements and simulation software elements; instead, flight software elements communicate with simulation software across flighl-like hardware interfaces (e.g., an 1553 bus or an RS232 serial line) or at least "black box" simulations of those interfaces. The intent is to minimize the number of instrumentation "scars" in the flight software, for two reasons: (1) to maximize the fidelity of flight simulations, and (2) to produce, as one effect of the development of the FST, a growing body of reusable, non-mission-specific flight software that could be used on future flight projects to reduce spacecraft development cost.

To serve this puI-pose, two distinct Tramel message spaces arc used in FST flight simulation: an "fsw" message space and an "sc" message space. Because message spaces are disjoint, flight software is guaranteed not to be dependent on simulation message exchange.

In the current "baseline" configuration, the flight software message space includes:

- a radio receiver interface node, which extracts command data from the radio's output bit stream;
- a simple executive node ("sequence manager"), which is responsible for executing command sequences;
- an on-board clock node ("time manager"), which publishes activity synchronization messages at 1-Hz and 10-Hz intervals;
- an attitude and articulation control system (AACS), comprising a half-dozen nodes, whit.1) computes estimated attitude from sensor output at 10-Hz intervals and activates thrusters to align actual spacecraft attitude with the commanded attitude;
- a power system interface node, which activates power system hardware elements in response to commands and issues power system telemetry messages;
- a radio system (RF) interface node, which controls the configuration of the RF hardware elements in response to commands and issues RF telemetry messages;
- a remote engineering unit (REU) interface node, which does analog-to-digital conversion of the data from a variety of engineering hardware subsystems and issues that data in engineering telemetry messages;
- a camera interface node, which activates photographic hardware in response to commands and issues the resulting images in telemetry messages;
- a telemetry management system, which collects telemetry (i.e., subscribes to telemetry messages), packages it according to CCSDS standards, and meters its release to the radio transmitter according to commanded downlink rules.

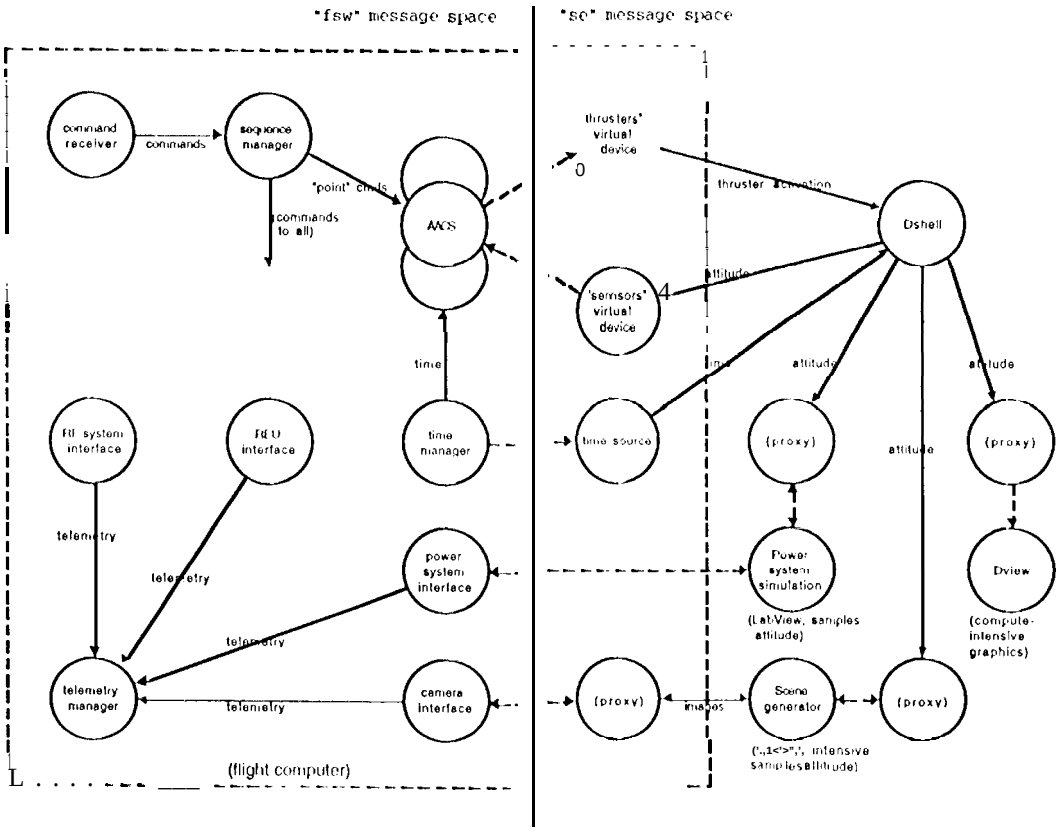

Figure 2: *Flight System Testbed software architecture*

The support equipment message space includes:

- a flight dynamics simulation node (Dshell) that models thrusters and gyms and continuously recomputes (normally at 10-1 Iz intervals) the attitude of the spacecraft as it spins in simulated space;
- a scene generation node that produces synthetic camera images with reference to spacecraft trajectory, time, spacecraft attitude, star catalogues, and known ephemerides;
- a power system simulation node, which computes power production from solar panels with reference to spacecraft attitude and known solar ephemerides;
- a graphical spacecraft attitude visualization node, named Dview, which displays an animated representation of the simulated spacecraft reflecting changes in the spacecraft's attitude over time.

Normally all of the fight software runs on a single VxWorks "target" processor so that the computational load on a flight computer can be accurately simulated. Support equipment software runs on a heterogeneous mix of VxWorks and UNIX platforms.

Also normal ly running on the flight processor arc two "virtual device" tasks that encapsulate hardware interfaces in black box fashion, simulating a 1553 bus, but that use Tramel to communicate with support equipment software: onc impersonates valve drive electronics, issuing thruster activation messages to Dshell; the other impersonates a gyro, receiving sensor reading messages from Dshell.

A third virtual device synchronizes the simulation environment with the flight machine's clock, echoing 10-Hz synchronization messages to Dshell. A Tramel proxy similarly links the camera interface node to tbc scene generator; in this case, the proxy serves as a "wormhole" between the "fsw" and "SC" message spaces, The power hardware simulation, scene generator, and graphical attitude visualizer arc all compute-intensive, so they interact with other "SC" message space nodes through Tramel proxies. The power interface node's access to the power hardware simulation is simulated in a TCP socket connection, but the REU's A-to-1) functions sample engineering circuitry directly. An RS232 serial cable links the RF interface node to the RF simulation systcm, which was developed before Tramel and has not yet been retrofitted as a node in the support equipment message space.

4. SPACE FLIGHT SIMULATION IN THE FST

Flight mission simulation software developed in the FST has used Tramel for event management and data communication since early 1995. Under Tramel, FST flight software running under VxWorks on a single-board computer can be commanded from a Tclshell running on a Silicon Graphics workstation and monitored by LabView "virtual instruments" on a Sparcstation, ancl all of these elements of software can be stopped, moved to different computers (even different types of computer, with different operating systems), and restarted -- while the application is running -- without alteration of any source code or configuration file text.

As an example of the flexibility afforded by the use of asynchronous message passing -- and, in particular, the publish/subscribe model -- as the organizing principle of the FS'l' software architecture, consider the attitude data produced by the fli.gilt dynamics simulation system.

The earliest usc of real-time attitude data from Dshell in the FST was as a critical source of feedback to the flight software's attitude control system: it enabled the

AACS and Dshell to run together in a closed loop that realistically tested the control law at the heart of attitude control.

The very same data, though, could be used to inform human users of the spacecraft's attitude just as readily as it informed the AACS: the Dview spacecraft visualization program, running on a Silicon Graphics workstation, could simply be an additional subscriber to the same messages. Likewise the scene generator running on a separate Spare machine could subscribe to the same spacecraft attitude data to determine what extent of the relevant star field to plot in a synthetic image.

Most recently, the power system simulation software (which is implemented in LabView) has been modified to subscribe to attitude data as well. This enables it to plot in real time the changes in power production from a solar panel due to changes in the angle between the plane of the panel and the direction of the sun with respect to the spacecraft. Such plots exactly track the spacecraft's oscillation within deadband as the attitude control system brings the spacecraft into alignment with a commanded attitude. Dview gives us an accurate alternative representation of this same oscillation -- a picture of a rotating spacecraft -- because it is driven by the same data.

The EST's RF simulation system will eventually be revised to use Tramel, so that it too can subscribe to spacecraft attitude data. That will enable the radio link simulation to compute signal strength and estimated bit error rate in real time, based on the angle between the antenna and the direction of Earth with respect to the spacecraft.

This incremental and internally consistent elaboration of the simulation environment was inexpensive to develop, in part because Tramel insulated the developers from most aspects of the configuration of the distributed application. The producers of the data could concentrate on what to produce rather than how to deliver it, and the consumers could think about how they would use the attitude data rather than how they could get it.

Moreover, the support equipment software elements are easy to operate in any number of run-time combinations because Tramel application configuration is developed dynamically and automatically, While the simulation is running, a new feature can be added to the scene generator and incorporated into the simulation -- or an altogether new consumer of attitude data can be added -- simply by starting the new program; no other element of the software is affected in any way.

4.1 Performance

Any system that attempts to provide application software with an abstract interface to transport protocols will incur some overhead, but Tramel is designed to minimize that overhead. Performance testing undertaken to date indicates that Tramel message transmission latency in an exchange of 128-byte messages between two VxWorks tasks running on a Heurikon HK V4F single-board computer (built on the Motorola M68040 processor), using a VxWorks pipe at the transport layer, averages somewhat under 700 microseconds; an indefinitely sustainable message exchange rate of 1200 to 1500 messages per second could reasonably be expected.

Naturally, message transmission rates are significantly different for different processors and transport protocols. Tramel message exchange rates on the order of 5000 128-byte messages per second have been observed between processes on a Silicon Graphics Indigo machine using TCP; between processes on a Spare Ultra and a Silicon Graphics Indigo, communicating over TCP on an Ethernet, typical rates are on the order of 3000 messages per second.

4.2 Problems

Version 3.0.1 of Tramel is currently in use in the FST. It addresses several application configuration problems that made earlier versions awkward to use, but the fundamental message exchange functionality has not required substantial revision since early 1995.

As noted earlier, not every type of application lends itself to the Tramel model of event-clrivcn, non-compute-intensive software, and not every developer is comfortable with the Tramel programming style (characterized by callback invocation from an event Imp). Identifying these mismatches and resolving them, in many cases by using the Tramel proxy system, is possibly the principal challenge in developing software under the Tramel asynchronous message passing scheme.

5. CONCLUSIONS

Asynchronous message passing, as implemented by Tramel, has simplified the development, integration, and operation of space flight simulation software in the Flight System Testbed. Performance has been adequate to satisfy the requirements of the applications involved, and the system has been flexible enough to accommodate a variety of software models.

6. PLANS FOR FUTURE WORK

A small number of mostly internal enhancemen ts to Tramel are planned for the end of 1996, but the bulk of Tramel application programming, functionality in now installed. Support for additional underlying transport protocols, such as UNIX sockets, will be added as needed. "1 looks" are in place to support the development of graphical utility software that will help programmers visualize, monitor, and debug a distributed application session.

The FST will be further exploiting the capabilities of Tramel as the Testbed software baseline itself grows:

- Residual non-Tramel interprocess communication links will gradually be phased out as time permits.
- Because the FST's reusable flight software is built on Tramel, integrating and testing additional capabilities such as these should be straightforward:
    - an autonomous, cm-board command sequence planner, which would issue command sequences to the sequence manager;
    - an autonomous optical navigation system, which would determine attitude and trajectory from images (published by the camera) and propose trajectory correction maneuvers to the planner;
    - an asteroid satellite detection system, which would identify possible asteroid satellites by examining sets of images taken over various illumination spectra and propose attitude adjustments (photo opportunities) to the planner.
- The FST is not the only customer for its reusable flight software. The Pluto Express project anti the Second-Generation Microspacecraft investigation already use this software, and other projects may adopt it in the future.

Broader use of Tramel is also possible:

- Given a socket-based radio link transport protocol Such as SCPS/TP (Ref. 3), Tramel could be used for communication between flight software elements and their counterparts on the ground. SCPS/TP developers have already demonstrated that SCPS/TP will operate across a simulated one-way light time delay of twenty minutes; 1<1'(~-based software might not perform acceptably under such conditions, but asynchronous message passing very likely would.
- Tramel has been proposed as a means of implementing real-time publication of annotations to space shuttle control mom documentation, as a part of the Johnson Space Flight Center's Electronic Document Preparation system.
- In general, Tramel could be considered for any computing problem whose solution might involve distributed and/or parallel processing.

7. REFERENCES

1. Rosenberry W, Kenney D, Fisher (i 1992, *Understanding DCE*, Sebastapol, Calif., O'Reilly

2. Object Management Group (OMG) 1995, *The common object request broker: Architecture and specification, revision 2.0*, ftp://ftp.omg.org/

3. Consultative Committee for Space Data Systems 1996, *Space Communications Protocol Standards (SCPS): SCPS Transport Protocol (SCPS-TP)*, CCSDS 714.0-W-1